Algorithms and Data Structures 2
CS 1501

Spring 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming deadlines:
  - Homework 5 due on 2/21
  - Lab 5 due on 2/25
  - Homework 6 due on 2/28
  - Assignment 1 due on 3/14
- Midterm exam on Wednesday 3/2
  - In-person, paper, closed book exam
- Faculty Candidate Talk tomorrow at 10:00 am at Sennott Square 5317
  - Topic: self-balancing binary search trees
  - very relevant talk to this class!

Previous lecture …

- Prefix-free Compression problem
  - Huffman coding as an optimal solution
  - Implementation details
    - storing the trie in the compressed file
    - Writing out variable-length bit strings

CourseMIRROR Reflections (interesting)

- I found it interesting how Huffman trees are created
- How to make a huffman tree
- I enjoyed going through examples
- It was quite fascinating how we were able to get back the original string after compression
- Manually compressing/decompressing data using Huffman trees
- How to reduce bits
- The most interesting part of class was amount of bits you save with huffman’s algorithm.
- Stepping through the Huffman compression algorithm and seeing the difference in bits once compressed

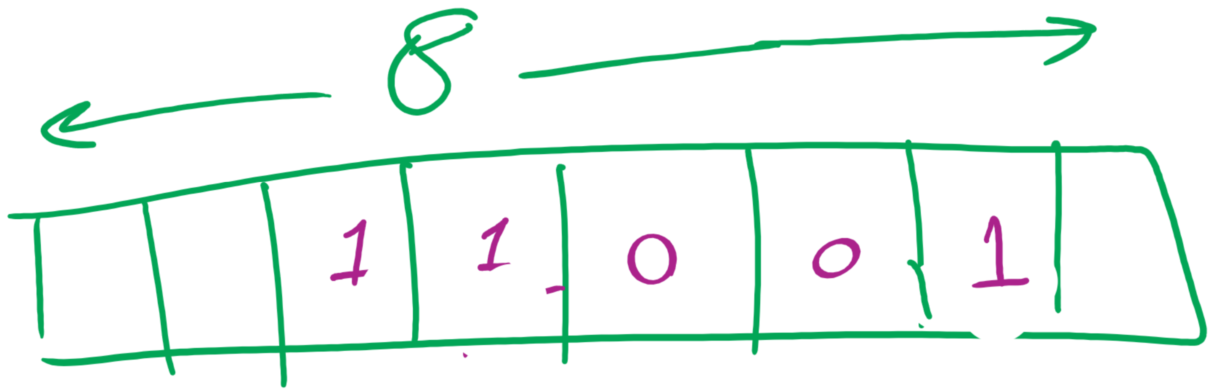
CourseMIRROR Reflections (confusing)

- Huffman compression algorithm steps
- Tree serialization/storage/encoding
- The difference between the compression algorithm and what's written to the file
- why to use an RST to implement a Huffman tree
- Why do we need to use a buffer to process bits?
- The most confusing part about class was the pairing of nodes in the new tree.
- The way to construct the forest in Huffmans compression can be a bit tricky as there are multiple ways to construct it

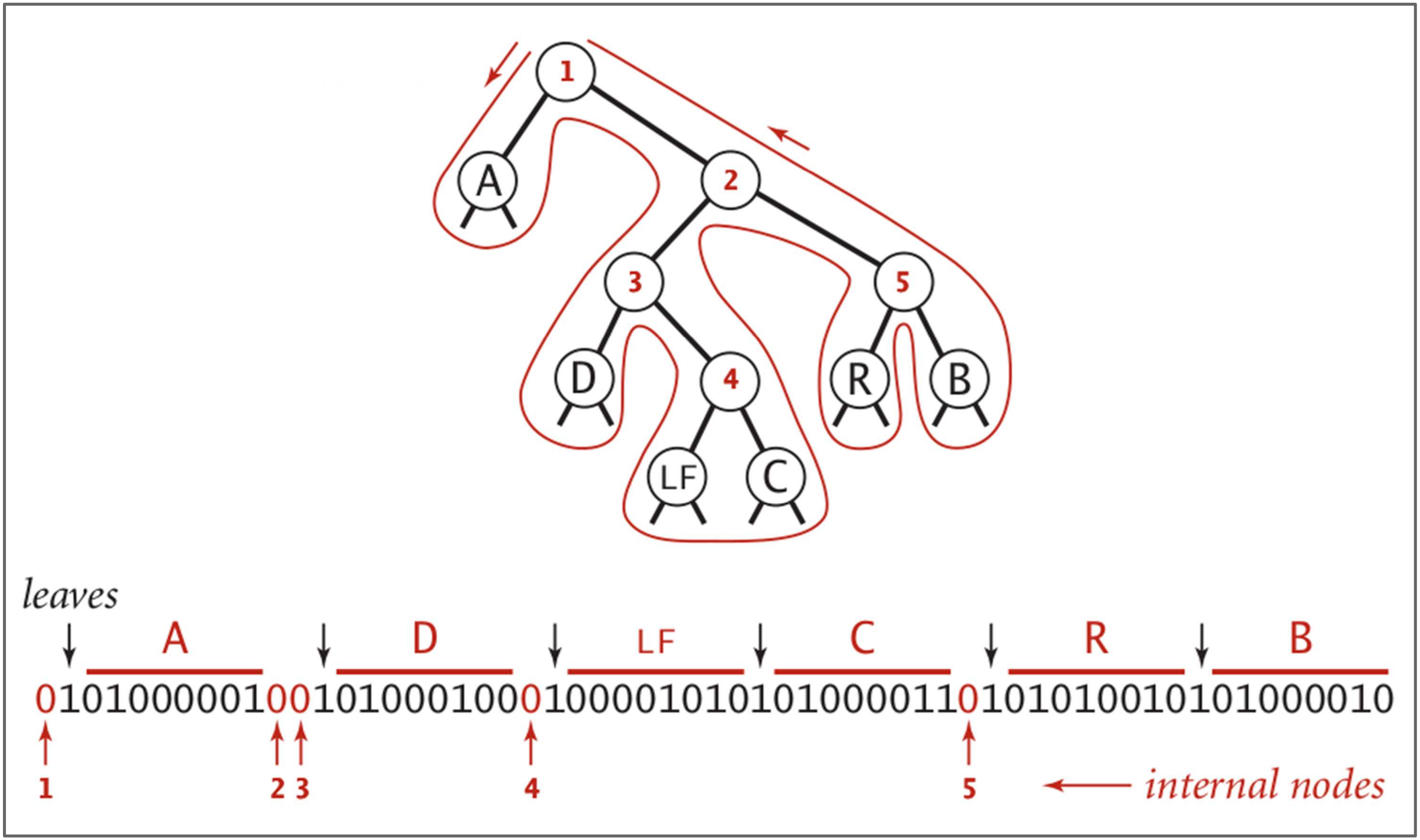
Binary I/O

Write

1
1
0
0
1
0
1
0



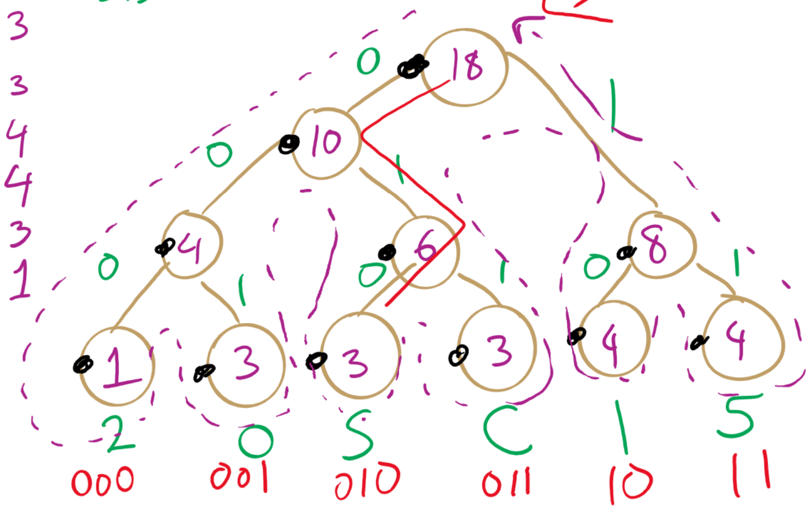
Representing tries as bitstrings



Huffman Compression Example

C : 3
S : 3
1 : 4
5 : 4
0 : 3
2 : 1

CS1501CS1502CS1550



C	011
S	010
1	10
5	11
0	001
2	000

C S 1 5 0 1 C S 1 5 0 2 C S 1 5 5 0

011 010 10 11 001 10 011 010 10 11 001 000 011 010 10 11 11 001

Trie 0001 ASCII for 2 1 ASCII for 0 1 ASCII for S - - - - -

CS1

Huffman pseudocode

- Encoding approach:
  - Read input
  - Compute frequencies
  - Build trie/codeword table
  - Write out trie as a bitstring to compressed file
  - Write out character count of input
  - Use table to write out the codeword for each input character
- Decoding approach:
  - Read trie
  - Read character count
  - Use trie to decode bitstring of compressed file

How do we determine character frequencies?

- Option 1: Preprocess the file to be compressed
  - Upside: Ensure that Huffman's algorithm will produce the best output for the given file
  - Downsides:
    - Requires two passes over the input, one to analyze frequencies/build the trie/build the code lookup table, and another to compress the file
    - Trie must be stored with the compressed file, reducing the quality of the compression
      - This especially hurts small files
      - Generally, large files are more amenable to Huffman compression
        - Just because a file is large, however, does not mean that it will compress well!

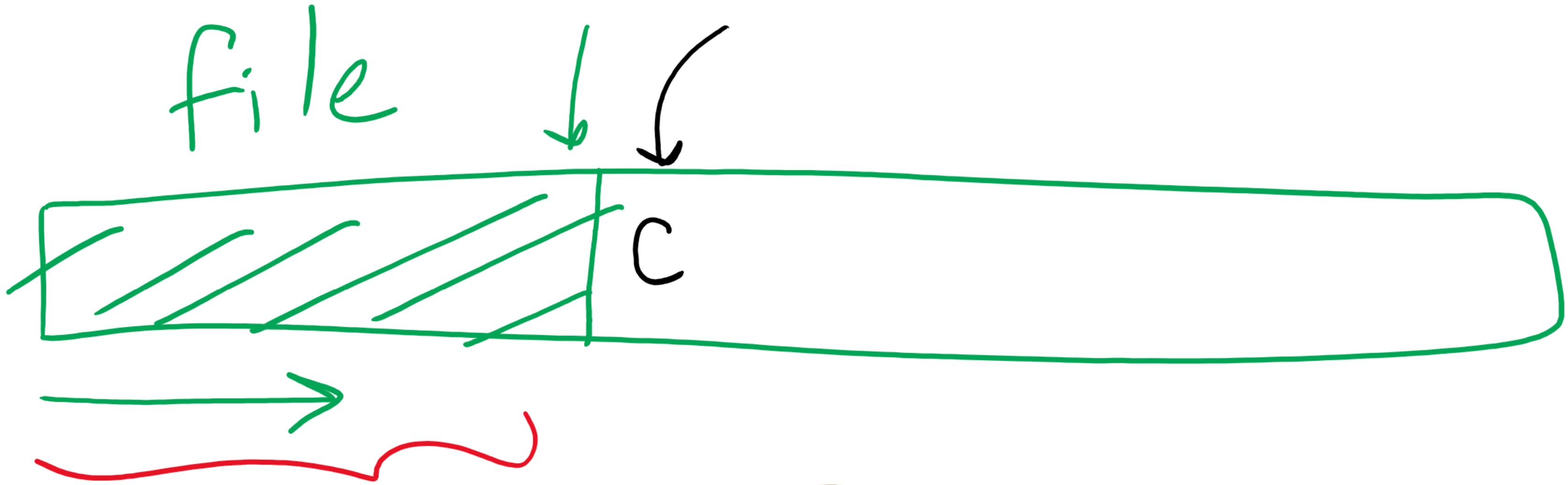
How do we determine character frequencies?

- Option 2: Use a static trie
  - Analyze multiple sample files, build a single tree that will be used for all compressions/expansions
  - Saves on trie storage overhead…
  - But in general not a very good approach
    - Different character frequency characteristics of different files means that a code set/trie that works well for one file could work very poorly for another
      - Could even cause an increase in file size after “compression”!

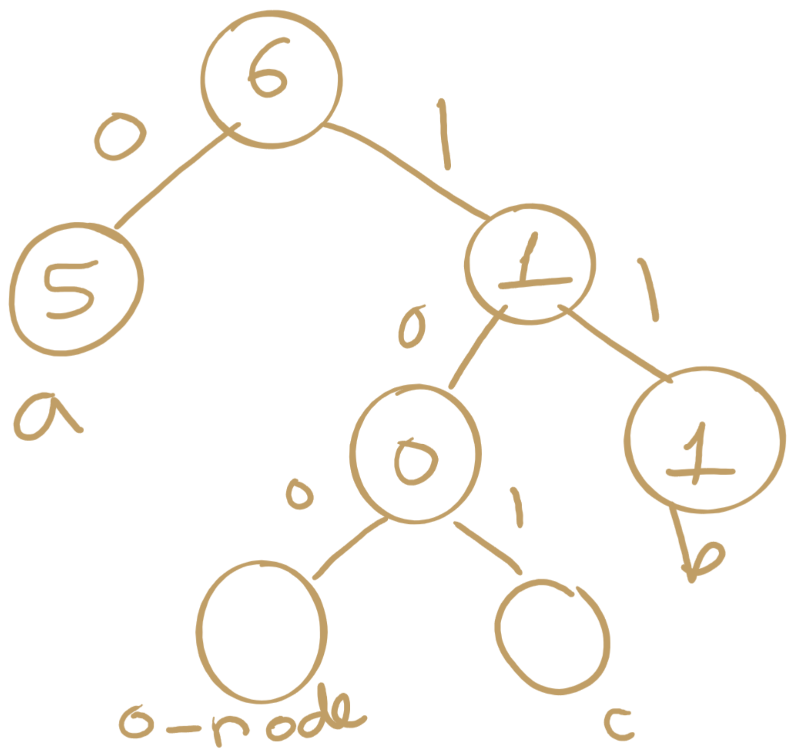
How do we determine character frequencies?

- Option 3: Adaptive Huffman coding
  - Single pass over the data to construct the codes and compress a file with no background knowledge of the source distribution
  - Not going to really focus on adaptive Huffman in the class, just pointing out that it exists...

Adaptive Huffman



a 5
b 1
0-node 0



Further implementation concerns

- Need to efficiently be able to select lowest weight trees to merge when constructing the trie
  - Can accomplish this using a *priority queue*
- Need to be able to read/write bitstrings!
  - Unless we pick multiples of 8 bits for our codewords, we will need to read/write fractions of bytes for our codewords
    - We're not actually going to do I/O on fraction of bytes
    - We'll maintain a buffer of bytes and perform bit processing on this buffer
    - See BinaryStdIn.java and BinaryStdOut.java

Ok, so how good is Huffman compression

- ASCII requires 8m bits to store m characters
- For a file containing c different characters
  - Given Huffman codes $\{h_0, h_1, h_2, \ldots, h_{(c-1)}\}$
  - And frequencies $\{f_0, f_1, f_2, \ldots, f_{(c-1)}\}$
  - Sum from 0 to c-1: $|h_i| * f_i$
- Total storage depends on the differences in frequencies
  - The bigger the differences, the better the potential for compression
- Huffman is optimal for character-by-character prefix-free encodings
  - Proof in Propositions T and U of Section 5.5 of the text

Problem of the Day

- Huffman's is optimal for character-by-character prefix-free encodings
  - Proof in Propositions T and U of Section 5.5 of the text
- But can we do better than Huffman's for lossless compression?

Problem of the Day: Lossless Compression

- Input: A sequence of characters
  - *n* characters
  - each encoded as an 8-bit Extended ASCII
- Output: A bit string
  - of length less than 8*n
  - the original sequence can be fully restored from the bitstring

Subproblem: Prefix-free Compression

- Input: A sequence of n characters
- Output: A codeword h_i for each character i such that
  - No codeword is a prefix of any other
  - When each character in the input sequence is replaced with each codeword
    - the length of that compressed sequence is minimum
    - the original sequence can be fully restored from the compressed bitstring

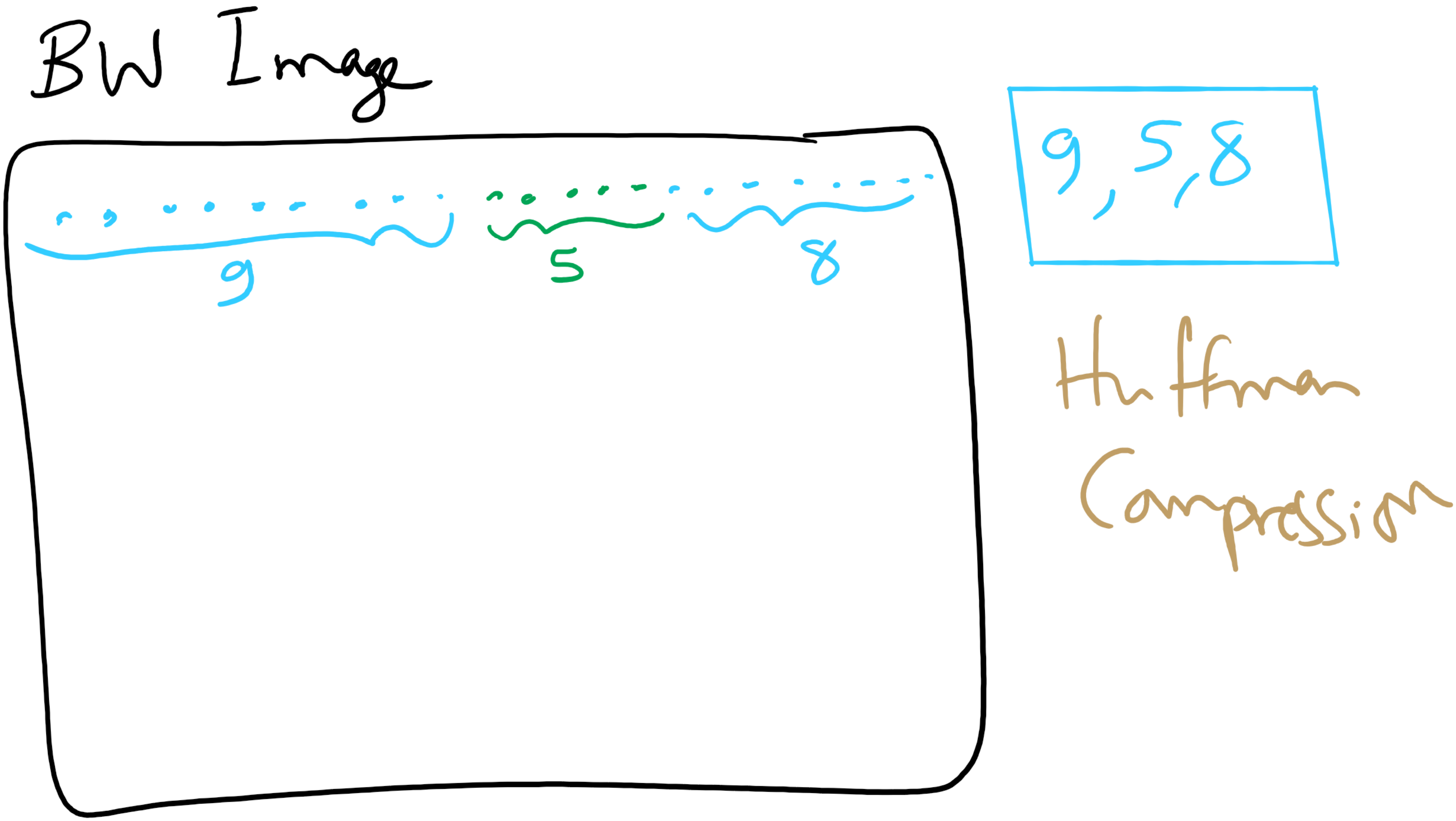
That seems like a bit of a caveat...

- Where does Huffman fall short?
  - What about repeated patterns of multiple characters?
    - Consider a file containing:
      - 1000 A's
      - 1000 B's
      - …
      - 1000 of every ASCII character
    - Will this compress at all with Huffman encoding?
      - Nope!
    - But it seems like it should be compressible…

Run length encoding

- Could represent the previously mentioned string as:
  - 1000A1000B1000C, etc.
    - Assuming we use 10 bits to represent the number of repeats, and 8 bits to represent the character…
      - 4608 bits needed to store run length encoded file
      - vs. 2048000 bits for input file
      - Huge savings!
- Note that this incredible compression performance is based on a very specific scenario…
  - Run length encoding is not generally effective for most files, as they often lack long runs of repeated characters

Run-length Encoding



Can we reason about how much a file can be compressed?

- Yes! Using Shannon Entropy

Information theory in a single slide...

- Founded by Claude Shannon in his paper "A Mathematical Theory of Communication"
- *Entropy* is a key measure in information theory
  - Slightly different from thermodynamic entropy
  - A measure of the unpredictability of information content
  - By losslessly compressing data, we represent the same information in less space
  - Hence, 8 bits of uncompressed text has less entropy than 8 bits of compressed data

Entropy Equation

$$\text{Entropy}(m) = -1 * \log_2 \Pr(m)$$

25%

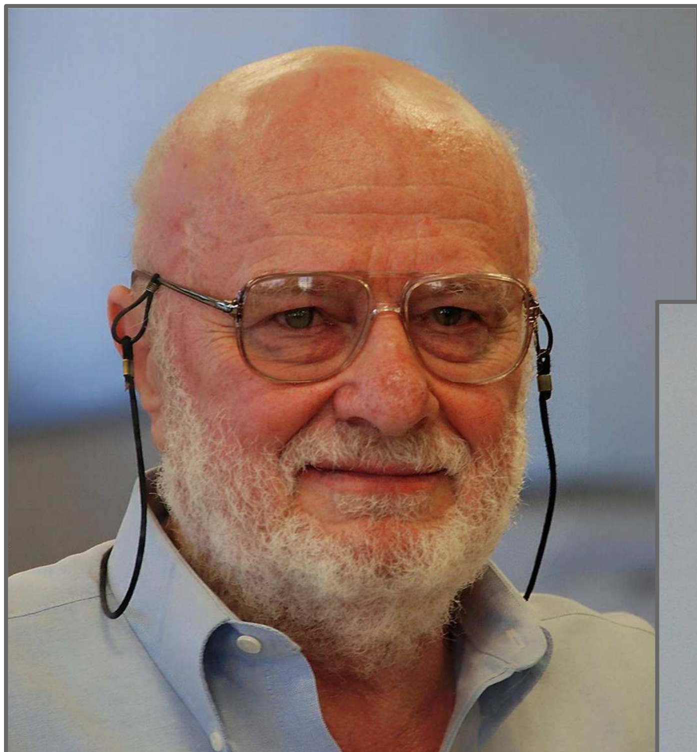
$$\text{Entropy}(c) = -1 * \log_2 \Pr(c)$$
$$= -1 * \log_2 \frac{1}{4}$$
$$= -1 * \log_2 2^{-2}$$
$$= -1 * -2 = 2 \text{ bits}$$

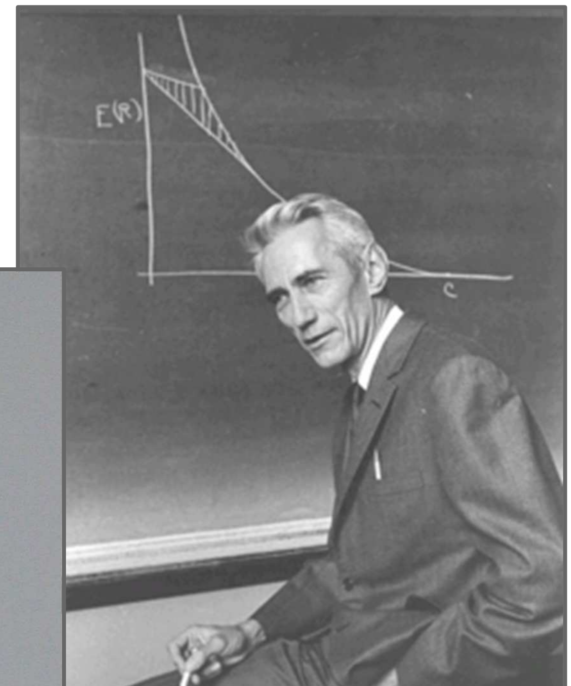
$$\frac{1}{2^{100}} \Rightarrow -1 * \log_2 2^{-100} = 100 \text{ bits}$$

Entropy applied to language:

- Translating a language into binary, the entropy is the average number of bits required to store a letter of the language
- Entropy of a message * length of message = amount of information contained in that message
- On average, a lossless compression scheme cannot compress a message to have more than 1 bit of information per bit of compressed message
- Uncompressed, English has between 0.6 and 1.3 bits of entropy per character of the message

What else can we do to compress files?





Patterns are compressible, need a general approach

- Huffman used variable-length codewords to represent fixed-length portions of the input…
  - Let's try another approach that uses fixed-length codewords to represent variable-length portions of the input
- Idea: the more characters can be represented in a single codeword, the better the compression
  - Consider "the": 24 bits in ASCII
  - Representing "the" with a single 12 bit codeword cuts the used space in half
    - Similarly, representing longer strings with a 12 bit codeword would mean even better savings!

How do we know that “the” will be in our file?

- Need to avoid the same problems as the use of a static trie for Huffman encoding…
- So use an adaptive algorithm and build up our patterns and codewords as we go through the file

LZW compression

- Initialize codebook to all single characters
  - e.g., character maps to its ASCII value
- While !EOF:
  - Match longest prefix in codebook
  - Output codeword
  - Take this longest prefix, add the next character in the file, and add the result to the dictionary with a new codeword

LZW compression example

- Compress, using 12 bit codewords:
  - TOBEORNOTTOBEORTOBEORNOT

Cur	Output	Add
T	84	TO:256
O	79	OB:257
B	66	BE:258
E	69	EO:259
O	79	OR:260
R	82	RN:261
N	78	NO:262
O	79	OT:263

T	84	TT:264
TO	256	TOB:265
BE	258	BEO:266
OR	260	ORT:267
TOB	265	TOBE:268
EO	259	EOR:269
RN	261	RNO:270
OT	263	--

Please submit your reflections by using the CourseMIRROR App

If you are having a problem with CourseMIRROR, please send an email to coursemirror.development@gmail.com